Programación Competitiva UFPS

Gerson Lázaro - Melissa Delgado

6 de mayo de 2016

Índice

1. Bonus: Input Output

1.1. scanf y printf

```
#include <cstdio>

scanf("%d",&value); //int
scanf("%ld",&value); //long y long int
scanf("%c",&value); //char
scanf("%f",&value); //float
scanf("%lf",&value); //double
scanf("%s",&value); //char*
scanf("%lld",&value); //long long int
scanf("%x",&value); //int hexadecimal
scanf("%o",&value); //int octal
```

2. Dynamic Programming

2.1. Knapsack

Dados N articulos, cada uno con su propio valor y peso y un tamaño máximo de una mochila, se debe calcular el valor maximo de los elementos que es posible llevar. Debe seleccionarse un subconjunto de objetos, de tal manera que quepan en la mochila y representen el mayor valor posible.

```
#include <algorithm>

const int MAX_WEIGHT = 40;//Peso maximo de la mochila
const int MAX_N = 1000; //Numero maximo de objetos
int N;//Numero de objetos
int prices[MAX_N];//precios de cada producto
int weights[MAX_N];//pesos de cada producto
int memo[MAX_N][MAX_WEIGHT];//tabla dp

//El metodo debe llamarse con 0 en el id, y la capacidad de la
    mochila en w
int knapsack(int id, int w) {
        if (id == N || w == 0) {
                return 0;
        }
        if (memo[id][w] != -1) {
                return memo[id][w];
        }
        if (weights[id] > w){
                memo[id][w] = knapsack(id + 1, w);
        }else{
                memo[id][w] = max(knapsack(id + 1, w), prices[id] +
                    knapsack(id + 1, w - weights[id]));
        }

        return memo[id][w];
}

//La tabla memo debe iniciar en -1
memset(memo, -1, sizeof memo);
```

2.2. Longest Increasing Subsequence

Halla la longitud de la subsecuencia creciente mas larga. MAX debe definirse en el tamaño limite del array, n es el tamaño del array. Puede aplicarse también sobre strings, cambiando el parametro int s[ ] por string s. Si debe ser estrictamente creciente, cambiar el $\leq$ de $s[j] \leq s[i]$ por $<$

```
const int MAX = 1005;
int memo[MAX];

int longestIncreasingSubsequence(int s[], int n){
        memo[0] = 1;
        int output = 0;
        for (int i = 1; i < n; i++){
                memo[i] = 1;
                for (int j = 0; j < i; j++){
                        if (s[j] <= s[i] && memo[i] < memo[j] + 1){
                                memo[i] = memo[j] + 1;
                        }
```

```
        }
        if(memo[i] > output){
               output = memo[i];
        }
    }
    return output;
}
```

2.3. Max Range Sum

Dada una lista de enteros, retorna la máxima suma de un rango de la lista.

```
#include <algorithm>

int maxRangeSum(vector<int> a){
    int sum = 0, ans = 0;
    for (int i = 0; i < a.size(); i++){
        if (sum + a[i] >= 0) {
            sum += a[i];
            ans = max(ans, sum);
        }else{
            sum = 0;
        }
    }
    return ans;
}
```

3. Geometry

3.1. Angle

Dados 3 puntos A, B, y C, determina el valor del angulo ABC (origen en B) en radianes. IMPORTANTE: Definir la estructura point y vec (Geometric Vector). Si se desea convertir a grados sexagesimales, revisar Sexagesimal degrees and radians.

```
#include <vector>
#include <cmath>

double angle(point a, point b, point c) {
    vec ba = toVector(b, a);
    vec bc = toVector(b, c);
    return acos((ba.x * bc.x + ba.y * bc.y) / sqrt((ba.x *
        ba.x + ba.y * ba.y) * (bc.x * bc.x + bc.y * bc.y)));
}
```

3.2. Area

Calcula el area de un polígono representado como un vector de puntos. IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polígono. El algorítmo utiliza el metodo de determinante de la matriz de puntos de la figura. IMPORTANTE: Debe definirse previamente la estructura point.

```
#include <vector>
#include <cmath>

double area(vector<point> P) {
    double result = 0.0, x1, y1, x2, y2;
    for (int i = 0; i < P.size()-1; i++) {
    x1 = P[i].x;
    x2 = P[i+1].x;
    y1 = P[i].y;
    y2 = P[i+1].y;
    result += ((x1 * y2) - (x2 * y1));
    }
    return fabs(result) / 2.0;
}
```

3.3. Collinear Points

Determina si el punto r está en la misma linea que los puntos p y q. IMPORTANTE: Deben incluirse las estructuras point y vec.

```
double cross(vec a, vec b) {
	return a.x * b.y - a.y * b.x;
}
bool collinear(point p, point q, point r) {
	return fabs(cross(toVector(p, q), toVector(p, r))) < 1e-9;
}
```

3.4. Convex Hull

Retorna el polígono convexo mas pequeño que cubre (ya sea en el borde o en el interior) un set de puntos. Recibe un vector de puntos, y retorna un vector de puntos indicando el polígono resultante. Es necesario que esten definidos previamente:

Estructuras: point y vec Métodos: collinear, euclideanDistance, inPolygon y angle.

```
#include <cmath>
#include <algorithm>
#include <vector>

point pivot;
bool angleCmp(point a, point b) {
	if (collinear(pivot, a, b)){
		return euclideanDistance(pivot, a) <
		    euclideanDistance(pivot, b);
	}

	double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
	double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
	return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}


vector<point> convexHull(vector<point> P) {
	int i, j, n = P.size();
	if (n <= 3) {
	if (!(P[0] == P[n-1])){
		P.push_back(P[0]);
	}
	return P;
	}
	int P0 = 0;
	for (i = 1; i < n; i++){
		if (P[i].y < P[P0].y || (P[i].y == P[P0].y &&
		    P[i].x > P[P0].x)){
		        P0 = i;
		}
	}

	point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
	pivot = P[0];
	sort(++P.begin(), P.end(), angleCmp);
	vector<point> S;
	S.push_back(P[n-1]);
	S.push_back(P[0]);
	S.push_back(P[1]);
	i = 2;
	while (i < n) {
	j = S.size()-1;
	if (ccw(S[j-1], S[j], P[i])){
		S.push_back(P[i++]);
	}else{
		S.pop_back();
	}
    }
	return S;
}
```

3.5. Euclidean Distance

Halla la distancia euclideana de 2 puntos en dos dimensiones (x,y). Para usar el primer método, debe definirse previamente la estructura point

```
#include <cmath>

/*Trabajando con estructuras de tipo punto*/
double euclideanDistance(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y);
}

/*Trabajando con los valores x y y de cada punto*/
double euclideanDistance(double x1, double y1, double x2, double
    y2){
  return hypot(x1 - x2, y1 - y2);
}
```

3.6. Geometric Vector

Dados dos puntos A y B, crea el vector $A->B$. IMPORTANTE: Debe definirse la estructura point. Es llamado vec para no confundirlo con el vector propio de c++.

```
struct vec {
        double x, y;
        vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVector(point a, point b) {
        return vec(b.x - a.x, b.y - a.y);
}
```

3.7. Perimeter

Calcula el perímetro de un polígono representado como un vector de puntos. IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polígono. La estructura point debe estar definida, al igual que el método euclideanDistance.

```
#include <vector>

double perimeter(vector<point> P) {
        double result = 0.0;
        for (int i = 0; i < P.size()-1; i++){
        result += euclideanDistance(P[i], P[i+1]);
        }
        return result;
}
```

3.8. Point in Polygon

Determina si un punto pt se encuentra en el polígono P. Este polígono se define como un vector de puntos, donde el punto 0 y n-1 son el mismo. IMPORTANTE: Deben incluirse las estructuras point y vec, ademas del método angle, y el método cross que se encuentra en Collinear Points.

```
#include <cmath>

bool ccw(point p, point q, point r) {
   return cross(toVector(p, q), toVector(p, r)) > 0;
}

bool inPolygon(point pt, vector<point> P) {
        if (P.size() == 0){
                return false;
        }
        double sum = 0;
        for (int i = 0; i < P.size()-1; i++) {
```

```
        if (ccw(pt, P[i], P[i+1])){
                sum += angle(P[i], pt, P[i+1]);
        }else{
                sum -= angle(P[i], pt, P[i+1]);
        }
    }
        return fabs(fabs(sum) - 2*acos(-1.0)) < 1e-9;
}
```

3.9. Point

La estructura punto será la base sobre la cual se ejecuten otros algoritmos.

```
#include <cmath>

struct point {
        double x, y;
        point() {
                x = y = 0.0;
        }
        point(double _x, double _y) : x(_x), y(_y) {}
        bool operator == (point other) const {
                return (fabs(x - other.x) < 1e-9 && (fabs(y -
                    other.y) < 1e-9));
        }
};
```

3.10. Sexagesimal degrees and radians

Conversiones de grados sexagesimales a radianes y viceversa.

```
#include <cmath>

double DegToRad(double d) {
        return d * acos(-1.0) / 180.0;
}

double RadToDeg(double r) {
        return r * 180.0 / acos(-1.0);
}
```

4. Graph

4.1. BFS

Algoritmo de búsqueda en anchura en grafos, recibe un nodo inicial s y visita todos los nodos alcanzables desde s. BFS también halla la distancia más corta entre el nodo inicial s y los demás nodos si todas las aristas tienen peso 1.

```
int v, e; //vertices, arcos
const int MAX=100005; //Cantidad maxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
long long dist[MAX]; //Estructura auxiliar para almacenar la
    distancia a cada nodo.

/*Debe llamarse al iniciar cada caso de prueba luego de haber
    leido v
Limpia todas las estructuras de datos.*/
static void init() {
  for (int j = 0; j <= v; j++) {
    dist[j] = -1;
    ady[j].clear();
  }
}

/*Este metodo se llama con el indice del nodo desde el que se
    desea comenzar
el recorrido.*/
static void bfs(int s){
  queue<int> q;
  q.push(s); //Inserto el nodo inicial
```

```
  dist[s] = 0;
  int actual, i, next;

  while( q.size() > 0 ){
    actual = q.front();
    q.pop();

    for( i = 0; i < ady[actual].size(); i++){
      next = ady[actual][i];
      if( dist[next] == -1 ){
        dist[next] = dist[actual] + 1;
        q.push(next);
      }
    }
  }

}
```

4.2. DFS

Algoritmo de búsqueda en profundidad para grafos. Parte de un nodo inicial s visita a todos sus vecinos. DFS puede ser usado para contar la cantidad de componentes conexas en un grafo y puede ser modificado para que retorne información de los nodos dependiendo del problema. Permite hallar ciclos en un grafo.

```
int v, e; //vertices, arcos
const int MAX=100005; //Cantidad maxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
int marked[MAX]; //Estructura auxiliar para marcar los nodos ya
    visitados

/*Debe llamarse al iniciar cada caso de prueba luego de haber
    leido v
Limpia todas las estructuras de datos.*/
void init(){
        for(int j = 0; j <= v; j++) {
    marked[j] = 0;
    ady[j].clear();
  }
}

/*Este metodo se llama con el indice del nodo desde el que se
    desea comenzar
el recorrido.*/
static void dfs(int s){
        marked[s] = 1;
        int i, next;

        for( i = 0; i < ady[s].size(); i++ ){
                next = ady[s][i];
                if( marked[next] == 0 ){
                        dfs(next);
                }
        }
}
```

4.3. Dijkstra's Algorithm

Algoritmo que dado un grafo con pesos no negativos halla la ruta mínima entre un nodo inicial s y todos los demás nodos.

```
#define Node pair<int,int>

int v,e; //v = cantidad de nodos, e = cantidad de aristas
const int MAX = 100001; //Cantidad Maxima de Nodos
vector<Node> ady[MAX]; //Lista de Adyacencia del grafo
int marked[MAX]; //Estructura auxiliar para marcar los nodos
    visitados
long long dist[MAX]; //Estructura auxiliar para llevar las
    distancias a cada nodo
int previous[MAX]; //Estructura auxiliar para almacenar las rutas

class cmp{
public:
  bool operator()(Node n1,Node n2){
```

```
    if(n1.second>n2.second)
      return true;
    else
      return false;
  }
};

/*Debe llamarse al iniciar cada caso de prueba para limpiar las
    estructuras.
Debe haberse leido v antes de hacer el llamado. */
void init(){
  long long max = LLONG_MAX;
  for(int j = 0; j <= v; j++){
    ady[j].clear();
    marked[j] = 0;
    previous[j] = -1;
    dist[j] = max;
  }
}

//El metodo debe llamarse con el indice del nodo inicial.
void dijkstra(int s){
    priority_queue< Node , vector<Node> , cmp > pq;
    pq.push(Node(s, 0));//se inserta a la cola el nodo Inicial.
    dist[s] = 0;
    int actual, j, adjacent;
    long long weight;

    while( !pq.empty() ){
      actual = pq.top().first;
      pq.pop();

      if( marked[actual] == 0 ){
        marked[actual] = 1;
        for( j = 0; j < ady[actual].size(); j++ ){
          adjacent = ady[actual][j].first;
          weight = ady[actual][j].second;
          if( marked[adjacent] == 0 ){
            if( dist[adjacent] > dist[actual] + weight ){
              dist[adjacent] = dist[actual] + weight;
              previous[adjacent] = actual;
              pq.push(Node( adjacent, dist[adjacent] ));
            }
          }
        }
      }
    }

}

int main(){
  int origen, destino;
  dijkstra(origen);
  //Para imprimir la distancia mas corta desde el nodo inicial al
      nodo destino
  dist[destino];

  //Para imprimir la ruta mas corta se debe imprimir de manera
      recursiva la estructura previous.
}
```

4.4. Flood Fill

Dado un grafo implicito colorea y cuenta el tamaño de las componentes conexas. Normalmente usado en rejillas 2D.

```
//aka Coloring the connected components

const int tam = 1000; //Maximo tamano de la rejilla
int dy[] = {1,1,0,-1,-1,-1, 0, 1}; //Estructura auxiliar para los
    desplazamientos
int dx[] = {0,1,1, 1, 0,-1,-1,-1};
char grid[tam][tam]; //Matriz de caracteres
int X, Y; //Tamano de la matriz

/*Este metodo debe ser llamado con las coordenadas x, y donde se
    inicia el
recorrido. c1 es el color que estoy buscando, c2 el color con el
    que se va
```

```
a pintar. Retorna el tamano de la componente conexa*/
int floodfill(int y, int x, char c1, char c2) {
	if (y < 0 || y >= Y || x < 0 || x >= X) return 0;

	if (grid[y][x] != c1) return 0; // base case

	int ans = 1;
	grid[y][x] = c2; // se cambia el color para prevenir ciclos

	for (int i = 0; i < 8; i++)
		ans += floodfill(y + dy[i], x + dx[i], c1, c2);

	return ans;
}
```

4.5. Floyd-Warshall's Algorithm

Algoritmo para grafos que halla la distancia mínima entre cualquier par de nodos. Matrix[i][j] guardará la distancia mínima entre el nodo i y el j.

```
#define Node pair<int,long long> //(Vertice adyacente, peso)

int v,e;
int matrix[505][505];

void floydWarshall(){
    int k=0;
    int aux, i ,j;

    while(k<v){
        for(i=0; i<v; i++){
            if(i!=k){
                for(j=0; j<v; j++){
                    if(j!=k){
                        aux=matrix[i][k]+matrix[k][j];
                        if(aux<matrix[i][j] && aux>0){
                            matrix[i][j]=aux;
                        }
                    }
                }
            }
        }
        k++;
    }
}
```

4.6. Kosaraju's Algorithm

Dado un grafo dirigido, calcula la componente fuertemente conexa a la que pertenece cada nodo. //aka Finding Strongly Connected Components

```
vector<int> ady[tam];
vector<int> rev[tam];
vector<int> topoSort;
int scc[tam];
int marked[tam];
int n,e; //vertices, arcos

void init(){
	topoSort.clear();
	for(int i=0; i<n; i++){
		ady[i].clear();
		marked[i]=0;
		scc[i]=-1;
		rev[i].clear();
	}
}

void topologicalSort(int u){
	int i, v;
	marked[u]=1;
	for(i=0; i<ady[u].size(); i++){
		v=ady[u][i];
		if(marked[v]==0)
			topologicalSort(v);
	}
```

```
        topoSort.push_back(u);
}

void dfs(int u, int comp){
        scc[u]=comp;
        int i, v;
        for(i=0; i<rev[u].size(); i++){
                v=rev[u][i];
                if(scc[v]==-1)
                        dfs(v, comp);
        }
}

int findScc(){
        int i, j, v;

        //Construye el grafo invertido
        for(i=0; i<n; i++){
                for(j=0; j<ady[i].size(); j++){
                        v=ady[i][j];
                        rev[v].push_back(i);
                }
        }

        //Enumera todos los nodos del grafo original
        for(i=0; i<n; i++){
                if(marked[i]==0)
                        topologicalSort(i);
        }

        reverse(topoSort.begin(), topoSort.end());

        //dfs, de acuerdo al orden del toposort
        int comp=0;
        for(int i=0; i<n; i++){
                v=topoSort[i];
                if(scc[v]==-1)
                        dfs(v, comp++);
        }
        return comp;
}
```

4.7. Kruskal's Algorithm

Algoritmo para hallar el arbol cobertor mínimo de un grafo no dirigido y conexo. Utiliza la técnica de Union-Find(Conjuntos disjuntos) para detectar que aristas generan ciclos. Para hallar los 2 arboles cobertores mínimos, se debe ejecutar el algoritmo v-1 veces, en cada una de ellas dscartar una de las aristas previamente elegidas en el arbol.

```
struct Edge{

  int source, dest, weight;

  bool operator != (const Edge& rhs) const{
    if(rhs.source != source || rhs.dest != dest || rhs.weight !=
        weight){
      return true;
    }
    return false;
  }

};

int v, e; //v = nodos, e = arcos
const int MAX = 10001; //Cantidad maxima de nodos
int parent[MAX]; //estructura de DS
int r[MAX]; //estructura de la implementacion de DS (rank)
Edge edges[MAX]; //Lista de arcos del grafo
Edge answer[MAX]; //Lista de arcos del arbol cobertor minimo

/*Debe llamarse al iniciar cada caso de prueba para limpiar las
    estructuras.
Debe haberse leido v antes de hacer el llamado. */
void init(){
  for(int i = 0; i < v; i++){
    parent[i] = i;
    r[i] = 0;
  }
}
```

```
int cmp(const void* a, const void* b){
  struct Edge* a1 = (struct Edge*)a;
  struct Edge* b1 = (struct Edge*)b;
  return a1->weight > b1->weight;
}

/*       Metodos Disjoint Set       */
int find(int i){
  if( parent[i] != i ){
    parent[i] = find(parent[i]);
  }
  return parent[i];
}

void unionFind(int x, int y){
  int xroot = find(x);
  int yroot = find(y);

  // Attach smaller r tree under root of high r tree
  if (r[xroot] < r[yroot])
    parent[xroot] = yroot;
  else if (r[xroot] > r[yroot])
    parent[yroot] = xroot;
  else{
    parent[yroot] = xroot;
    r[xroot]++;
  }

}

/*      FIN: Metodos Disjoint Set      */

/*El arbol cobertor minimo del grafo queda almacenado en el
vector de arcos answer*/
void kruskall(){
  Edge actual;
  int aux = 0;
  int i = 0;
  int x, y;
  qsort(edges, e, sizeof(edges[0]), cmp);
  while(aux < v-1){
    actual = edges[i];
    x = find(actual.source);
    y = find(actual.dest);

    if(x != y){
      answer[aux] = actual;
      aux++;
      unionFind(x, y);
    }
    i++;
  }
}

int main(){
  int s, d, w;
  //Los arcos se inicializan asi
  edges[i].source = s;
  edges[i].dest = d;
  edges[i].weight = w;

  kruskall();

}
```

4.8. Maxflow

Dado un grafo, halla el máximo flujo entre una fuente s y un sumidero t.

```
vector<int> adyNetwork [105];
int capacity [105] [105]; //Capacidad de aristas de la red
int flow [105] [105]; //Flujo de cada arista
int anterior [105];

void connect(int i, int j, int cap){
    adyNetwork[i].push_back(j);
    adyNetwork[j].push_back(i);
```

```
    capacity[i][j]+=cap;
    //Si el grafo es dirigido no hacer esta linea
    //capacity[j][i]+=cap;
}

int maxflow(int s, int t, int n){ //s=fuente, t=sumidero,
    n=numero de nodos
    int i, j, maxFlow, u, v, extra, start, end;
    for(i=0; i<=n; i++){
        for(j=0; j<=n; j++){
            flow[i][j]=0;
        }
    }

    maxFlow=0;

    while(true){
        for(i=0; i<=n; i++) anterior[i]=-1;

        queue<int> q;
        q.push(s);
        anterior[s]=-2;

        while(q.size()>0){
            u=q.front();
            q.pop();
            if(u==t) break;
            for(j=0; j<adyNetwork[u].size(); j++){

                v=adyNetwork[u][j];
                if(anterior[v]==-1 && capacity[u][v] -
                    flow[u][v]>0){
                    q.push(v);
                    anterior[v]=u;
                }
            }
        }
        if(anterior[t]==-1)break;

        extra=1<<30;
        end=t;
        while(end!=s){
            start=anterior[end];
            extra=min(extra, capacity[start][end]-flow[start][end]);
            end=start;
        }

        end=t;
        while(end!=s){
            start=anterior[end];
            flow[start][end]+=extra;
            flow[end][start]= -flow[start][end];
            end=start;
        }

        maxFlow+=extra;
    }

    return maxFlow;
}

int main(){
    //Para cada arista
    connect(s,d,f); //origen, destino, flujo
}
```

4.9. Tarjan's Algorithm

Algoritmo para hallar los puentes e itsmos en un grafo no dirigido.

```
vector<int> ady[1010];
int marked[1010];
int previous[1010];
int dfs_low[1010];
int dfs_num[1010];
int itsmos[1010];
int n, e;
int dfsRoot,rootChildren,cont;
vector<pair<int,int>> bridges;
```

```
void init(){
    bridges.clear();
    cont=0;
    int i;
    for(i=0; i<n; i++){
        ady[i].clear();
        marked[i]=0;
        previous[i]=-1;
        itsmos[i]=0;
    }
}

void dfs(int u){
    dfs_low[u]=dfs_num[u]=cont;
    cont++;
    marked[u]=1;
    int j, v;

    for(j=0; j<ady[u].size(); j++){
        v=ady[u][j];
        if(marked[v]==0){
            previous[v]=u;
            //para el caso especial
            if(u==dfsRoot){
                rootChildren++;
            }
            dfs(v);
            //Itsmos
            if(dfs_low[v]>=dfs_num[u]){
                itsmos[u]=1;
            }
            //Bridges
            if(dfs_low[v]>dfs_num[u]){
                bridges.push_back(make_pair(min(u,v),max(u,v)));
            }
            dfs_low[u]=min(dfs_low[u], dfs_low[v]);
        }else if(v!=previous[u]){ //Arco que no sea backtrack
            dfs_low[u]=min(dfs_low[u], dfs_num[v]);
        }
    }
}

int main(){
    //Antes de ejecutar el Algoritmo
    cont=0;
    dfsRoot=0;
    rootChildren=0;
    dfs(0);
}
```

4.10. Topological Sort

Dado un grafo acíclico y dirigido, ordena los nodos linealmente de tal manera que si existe una arista entre los nodos u y v entonces u aparece antes que v. Este ordenamiento es una manera de poner todos los nodos en una línea recta de tal manera que las aristas vayan de izquierda a derecha.

```
int v;
vector<int> topoSort;
vector<int> ady[tam];
int marked[tam];

void init(){
       for (int j = 0; j <= v; j++) {
      marked[j] = 0;
      ady[j].clear();
    }
    topoSort.clear();
}

void dfs(int u){
       int i, v;
       marked[u]==1;
       for(i=0; i<ady[u].size(); i++){
               v=ady[u][i];
               if(marked[v]==0)
                       dfs(v);
       }
```

```
        topoSort.push_back(u);
}

int main(){
        init();
        int i;
        for(i=0; i<v; i++){
                if(marked[i]==0)
                        dfs(i)
        }
        //imprimir topoSort en reversa :3
}
```

5. Math

5.1. Binary Exponentiation

Realiza a^b y retorna el resultado módulo c. Si se elimina el módulo c, debe tenerse precaución para no exceder el límite

```
int binaryExponentiation(int a, int b, int c){
    if (b == 0){
        return 1;
    }
    if (b % 2 == 0){
        int temp = binaryExponentiation(a,b/2, c);
        return ((long long)(temp) * temp) % c;
    }else{
        int temp = binaryExponentiation(a, b-1, c);
        return ((long long)(temp) * a) % c;
    }
}
```

5.2. Binomial Coefficient

Calcula el coeficiente binomial nCr, entendido como el número de subconjuntos de k elementos escogidos de un conjunto con n elementos.

```
long long binomialCoefficient(long long n, long long r) {
  if (r < 0 || n < r) {
        return 0;
  }
  r = min(r, n - r);
  long long ans = 1;
  for (int i = 1; i <= r; i++) {
    ans = ans * (n - i + 1) / i;
  }
  return ans;
}
```

5.3. Catalan Number

Guarda en el array Catalan Numbers los numeros de Catalan hasta MAX.

```
const int MAX = 30;
long long catalanNumbers[MAX+1];

void catalan(){
        catalanNumbers[0] = 1;
        for(int i = 1; i <= MAX; i++){
                catalanNumbers[i] = (long
                    long)(catalanNumbers[i-1]*((double)(2*((2 * i)-
                    1))/(i + 1)));
        }
}
```

5.4. Euler Totient

Función totient o indicatriz (ϕ) de Euler. Para cada posición n del array result retorna el número de enteros positivos menores o iguales a n que son coprimos con n (Coprimos: MCD = 1)

```
#include <string.h>

const int MAX = 100;
int result[MAX];

void totient () {
	bool temp[MAX];
	int i,j;
	memset(temp,1,sizeof(temp));
	for(i = 0; i < MAX; i++) {
		result[i] = i;
	}
	for(i = 2; i < MAX; i++){
		if(temp[i]) {
			for(j = i; j < MAX ; j += i){
				temp[j] = false;
				result[j] = result[j] - (result[j]/i)
				    ;
			}
			temp[i] = true ;
		}
	}
}
```

5.5. Gaussian Elimination

Resuelve sistemas de ecuaciones lineales por eliminación Gaussiana. matrix contiene los valores de la matriz cuadrada y result los resultados de las ecuaciones. Retorna un vector con el valor de las n incongnitas. Los resultados pueden necesitar redondeo.

```
#include <vector>
#include <algorithm>
#include <limits>
#include <cmath>

const int MAX = 100;
int n = 3;
double matrix[MAX][MAX];
double result[MAX];

vector<double> gauss() {

	vector<double> ans(n, 0);
	double temp;
	for (int i = 0; i < n; i++) {
	int pivot = i;
	    for (int j = i + 1; j < n; j++) {
	        temp = fabs(matrix[j][i]) - fabs(matrix[pivot][i]);
	        if (temp > numeric_limits<double>::epsilon()) {
	            pivot = j;
	        }
	    }

	    swap(matrix[i], matrix[pivot]);
	    swap(result[i], result[pivot]);

	    if (!(fabs(matrix[i][i]) <
	        numeric_limits<double>::epsilon())) {

	        for (int k = i + 1; k < n; k++) {
	            temp = -matrix[k][i] / matrix[i][i];
	            matrix[k][i] = 0;
	            for (int l = i + 1; l < n; l++) {
	                matrix[k][l] += matrix[i][l] * temp;
	            }
	            result[k] += result[i] * temp;
	        }
	    }
	}
	for (int m = n - 1; m >= 0; m--) {
```

```
        temp = result[m];
        for (int i = n - 1; i > m; i--) {
                temp -= ans[i] * matrix[m][i];
        }
        ans[m] = temp / matrix[m][m];
        }
        return ans;
}
```

5.6. Greatest common divisor

Calcula el máximo común divisor entre a y b mediante el algoritmo de Euclides

```
int mcd(int a, int b) {
        int aux;
        while(b!=0){
                a %= b;
                aux = b;
                b = a;
                a = aux;
        }
        return a;
}
```

5.7. Lowest Common Multiple

Calculo del mínimo común múltiplo usando el máximo común divisor. REQUIERE mcd(a,b)

```
int mcm(int a, int b) {
        return a*b/mcd(a,b);
}
```

5.8. Miller-Rabin

La función de Miller-Rabin determina si un número dado es o no un número primo.

```
#include <cstdlib>

long long mulmod(long long a, long long b, long long mod){
    long long x = 0;
    long long y = a % mod;
    while (b > 0){
        if (b % 2 == 1){
            x = (x + y) % mod;
        }
        y = (y * 2) % mod;
        b /= 2;
    }
    return x % mod;
}

long long modulo(long long base, long long exponent, long long
    mod){
    long long x = 1;
    long long y = base;
    while (exponent > 0){
        if (exponent % 2 == 1)
            x = (x * y) % mod;
        y = (y * y) % mod;
        exponent = exponent / 2;
    }
    return x % mod;
}


bool miller(long long p){
    if (p < 2){
        return false;
    }
    if (p != 2 && p % 2==0){
        return false;
```

```
    }
    long long s = p - 1;
    while (s % 2 == 0){
        s /= 2;
    }
    for (int i = 0; i < 5; i++){
        long long a = rand() % (p - 1) + 1;
        long long temp = s;
        long long mod = modulo(a, temp, p);
        while (temp != p - 1 && mod != 1 && mod != p - 1){
            mod = mulmod(mod, mod, p);
            temp *= 2;
        }
        if (mod != p - 1 && temp % 2 == 0){
            return false;
        }
    }

    return true;
}
```

5.9. Prime Factorization

Guarda en primeFactors la lista de factores primos del value de menor a mayor. IMPORTANTE: Debe ejecutarse primero la criba de Eratostenes. La criba debe existir al menos hasta la raiz cuadrada de value.

```
#include <vector>

vector <long long> primeFactors;

void calculatePrimeFactors(long long value){
        primeFactors.clear();
        long long temp = value;
        int factor;
        for (int i = 0; (long long)primes[i] * primes[i] <= value;
            ++i){
                factor = primes[i];
                while (temp % factor == 0){
                        primeFactors.push_back(factor);
                        temp /= factor;
                }
        }
        if (temp != 1) {
                primeFactors.push_back(temp);
        }
}
```

5.10. Sieve of Eratosthenes

Guarda en primes los números primos menores o iguales a MAX

```
#include <vector>

const int MAX = 10000000;
vector<int> primes;
bool sieve[MAX+5];

void calculatePrimes() {
  sieve[0] = sieve[1] = 1;
  int i;
  for (i = 2; i * i <= MAX; i++) {
    if (!sieve[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= MAX; j += i)
        sieve[j] = true;
    }
  }
  for(;i <= MAX; i++){
      if (!sieve[i]) {
    primes.push_back(i);
      }
  }
}
```

6. String

6.1. KMP's Algorithm

Encuentra si el string pattern se encuentra en el string cadena.

```
#include <vector>

vector<int> table(string pattern){
	int m=pattern.size();
	vector<int> border(m);
	border[0]=0;

	for(int i=1; i<m; ++i){
		border[i]=border[i-1];
		while(border[i]>0 &&
		    pattern[i]!=pattern[border[i]]){
			border[i]=border[border[i]-1];
		}
		if(pattern[i] == pattern[border[i]]){
			border[i]++;
		}
	}
	return border;
}

bool kmp(string cadena, string pattern){
	int n=cadena.size();
	int m=pattern.size();
	vector<int> tab=table(pattern);
	int seen=0;

	for(int i=0; i<n; i++){
		while(seen>0 && cadena[i]!=pattern[seen]){
			seen=tab[seen-1];
		}
		if(cadena[i]==pattern[seen])
			seen++;
		if(seen==m){
			return true;
		}
	}
	return false;
}
```

7. Tips and formulas

7.1. ASCII Table

Caracteres ASCII con sus respectivos valores numéricos.

No.	ASCII	No.	ASCII
0	NUL	16	DLE
1	SOH	17	DC1
2	STX	18	DC2
3	ETX	19	DC3
4	EOT	20	DC4
5	ENQ	21	NAK
6	ACK	22	SYN
7	BEL	23	ETB
8	BS	24	CAN
9	TAB	25	EM
10	LF	26	SUB
11	VT	27	ESC
12	FF	28	FS
13	CR	29	GS
14	SO	30	RS
15	SI	31	US

No.	ASCII	No.	ASCII
32	(space)	48	0
33	!	49	1
34	”	50	2
35	#	51	3
36	$	52	4
37	%	53	5

38	&	54	6
39	’	55	7
40	(	56	8
41	)	57	9
42	*	58	:
43	+	59	;
44	,	60	¡
45	-	61	=
46	.	62	¿
47	/	63	?

No.	**ASCII**	**No.**	**ASCII**
64	@	80	P
65	A	81	Q
66	B	82	R
67	C	83	S
68	D	84	T
69	E	85	U
70	F	86	V
71	G	87	W
72	H	88	X
73	I	89	Y
74	J	90	Z
75	K	91	[
76	L	92	\
77	M	93	]
78	N	94	ˆ
79	O	95	_

No.	**ASCII**	**No.**	**ASCII**
96	‘	112	p
97	a	113	q
98	b	114	r
99	c	115	s
100	d	116	t
101	e	117	u
102	f	118	v
103	g	119	w
104	h	120	x
105	i	121	y
106	j	122	z
107	k	123	{
108	l	124	|
109	m	125	}
110	n	126	˜
111	o	127	

7.2. Catalan Number

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

Primeros 30 números de Catalán:

n	C_n
0	1
1	1
2	2
3	5
4	14
5	42
6	132
7	429
8	1.430
9	4.862
10	16.796
11	58.786
12	208.012
13	742.900
14	2.674.440

15	9.694.845
16	35.357.670
17	129.644.790
18	477.638. 700
19	1.767.263.190
20	6.564.120.420
21	24.466.267.020
22	91.482.563.640
23	343.059.613.650
24	1.289.904.147.324
25	4.861.946.401.452
26	18.367.353.072.152
27	69.533.550.916.004
28	263.747.951.750.360
29	1.002.242.216.651.368
30	3.814.986.502.092.304

7.3. Euclidean Distance

$$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

7.4. Permutation and combination

Combinación (Coeficiente Binomial): Número de subconjuntos de k elementos escogidos de un conjunto con n elementos

$$\binom{n}{k} = \binom{n}{n-k} = \frac{n!}{k!(n-k)!}$$

Combinación con repetición: Número de grupos formados por n elementos, partiendo de m tipos de elementos.

$$CR_m^n = \binom{m+n-1}{n} = \frac{(m+n-1)!}{n!(m-1)!}$$

Permutación: Número de formas de agrupar n elementos, donde importa el orden y sin repetir elementos

$$P_n = n!$$

Elegir r elementos de n posibles con repetición

$$n^r$$

Permutaciones con repetición: Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ...

$$PR_n^{a,b,c...} = \frac{P_n}{a!b!c!...}$$

Permutaciones sin repetición: Número de formas de agrupar r elementos de n disponibles, sin repetir elementos

$$\frac{n!}{(n-r)!}$$

7.5. Time Complexities

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algoritmicas. Tomar esta tabla solo como referencia.

Complexity	**n**
$O(n!)$	11
$O(n^5)$	50
$O(2^n * n^2)$	18
$O(2^n * n)$	22
$O(n^4)$	100
$O(n^3)$	500
$O(n^2 \log_2 n)$	1.000
$O(n^2)$	10.000
$O(n \log_2 n)$	10^6

$O(n)$	10^8
$O(\sqrt{n})$	10^{16}
$O(\log_2 n)$	-
$O(1)$	-

7.6. mod: properties

1. (a % b) % b = a % b (Propiedad neutro)
2. (ab) % c = ((a % c)(b % c)) % c (Propiedad asociativa en multiplicación)
3. (a + b) % c = ((a % c) + (b % c)) % c (Propiedad asociativa en suma)